Backend as a Service: Neues Geschäftsfeld für die Cloud ▸64

S&S

JAVA Mag

11.2013

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Tomcat 8
Erste Rauchzeichen der neuen Version ▸14

NoSQL mit Cassandra
Skalierungssieger auf der ganzen Linie ▸48

Spring Batch 2.2
Java Config und Spring-Data-Support ▸78

Continuous Delivery

W-JAX 13
Alle Infos hier im Heft!
▸35

Prinzipien und

Sonderdruck für
www.codecentric.de

codecentric

Vor- und Nachteile in der Praxis ▸42

Java Config, Spring-Data-Support und Co.

Spring Batch 2.2

NEUE SERIE

Es tut sich einiges in der Batchverarbeitung mit Java – Grund genug für den Start einer dreiteiligen Artikelserie. Hier geht es zunächst um das neue Spring-Batch-Release 2.2.0/2.2.1. Welche Features kommen hinzu, wie sind sie zu bewerten? In der nächsten Ausgabe schauen wir uns den JSR-352 – Batch Applications for the Java Platform – genauer an. Nach fast zweijähriger Entwicklung ist die Spezifikation nun final und Teil von Java EE 7. Und dann gibt es da noch Spring XD, ein neues Projekt, das sich zum Ziel gesetzt hat, die Entwicklung von Big-Data-Anwendungen zu vereinfachen, und dazu auf Spring Batch und Spring Integration aufsetzt.

von Tobias Flohre

Spring Batch ist das führende Open-Source-Framework für die Batchverarbeitung in Java. Einführende Artikel gab es bereits im Java Magazin [1], [2]. Hilfreich ist auch die Projektdokumentation [3]. Mit dem neuen Project Lead Michael Minella (ebenfalls in der Expert Group zum JSR-352) nahm die Entwicklung am Framework Ende letzten Jahres wieder Fahrt auf. Im Juni kam dann die Version 2.2.0, im Juli bereits Version 2.2.1.

Die zwei wichtigsten Neuerungen betreffen zwei starke Entwicklungen der letzten Jahre im Spring-Ökosystem, die jetzt auch in Spring Batch verfügbar sind. Das ist zum einen die Möglichkeit, Konfigurationen in Java zu erstellen und kein XML mehr zu benötigen. Zum ersten Mal Ende 2009 in Spring 3.0 verfügbar, entwickelt es sich immer mehr zum Standard für Spring-Anwendungen. Vorteile dieser Art der Konfiguration waren auch schon Thema im Java Magazin [4]. Die andere starke Entwicklung ist das Spring-Data-Projekt, das den Zugriff auf NoSQL-Stores vereinheitlicht und vereinfacht [5]. Spring Batch bietet nun einige konkrete *ItemReader*/*ItemWriter*-Implementierungen sowie eine generische, auf Spring Datas Repository-Abstraktion basierende Implementierung. Die dritte wichtige Änderung ist die Einführung von nicht identifizierenden Jobparametern, da dieses Feature eine Datenbankänderung mit sich bringt.

Konfiguration in Java

Bei der Java-basierten Konfiguration geht es darum, die zentralen XML-Konfigurationsdateien durch zentrale Java-Konfigurationsklassen zu ersetzen. Da in diesen Konfigurationsklassen auch viel mit Annotationen gearbeitet wird, erlebe ich häufig, dass dieser Konfigurationsansatz mit der annotationsbasierten Konfiguration verwechselt wird. Der elementare Unterschied ist jedoch, dass wir bei der Java-basierten Konfiguration weiterhin eine zentrale Konfiguration haben, während sich diese bei der annotationsbasierten Konfiguration dezentralisiert in den Fachklassen befindet. Für einen Vergleich der drei Konfigurationsvarianten ohne Bezug auf Spring Batch gab es bereits einen Artikel im Java Magazin [4].

Wie in einem guten Design üblich, trennen wir bei den folgenden Beispielen die Konfiguration der Infrastruktur von der Konfiguration der Jobs. Damit wir die Infrastrukturkonfiguration leichter austauschen können, definiere ich über ein Interface die von unterschiedlichen Implementierungen zu liefernden Komponenten, in diesem Fall eine *DataSource*:

```
public interface InfrastructureConfiguration {

  public abstract DataSource dataSource();

}
```

Warum ist eine Austauschbarkeit der Infrastrukturkonfiguration überhaupt wichtig? In der Regel hat man mindestens zwei unterschiedliche Implementierungen: eine für den Fall, dass wir den Job in einem JUnit Test laufen lassen wollen und eine weitere für den Betrieb im Container. Schauen wir uns zunächst die Implementierung für den Testbetrieb an (Listing 1).

Mit der Annotation *@Configuration* wird diese Klasse als Spring-Konfigurationsklasse markiert. Jede Methode, die mit *@Bean* annotiert ist, liefert eine Bean für den *ApplicationContext*. In unserem Fall ist das eine *DataSource*, die auf eine In-Memory-HSQL-Daten-

Artikelserie

Teil 1: Java Config, Spring-Data-Support und Co.
Teil 2: JSR-352 – Batch Applications for the Java Platform
Teil 3: Spring XD – Ordnung für Big-Data-Datenströme

bank zeigt, die mit zwei Skripten initialisiert wurde: einmal für die Spring-Batch-Metadaten und einmal für Fachdaten.

Wer Spring Batch kennt, der weiß, dass die Minimalinfrastruktur für den Betrieb von Jobs aus einem *JobRepository*, einem *JobLauncher* und einem *PlatformTransactionManager* besteht. Wo befinden sich diese in der Konfiguration in Listing 1? Die Annotation *@EnableBatchProcessing* sorgt automatisch dafür, dass diese erzeugt werden. Es wird ein *DataSourceTransactionManager* erstellt, der auf der *DataSource* operiert, ein *JobRepository*, das den *DataSourceTransactionManager* und die *DataSource* nutzt, und ein *JobLauncher*, der das *JobRepository* verwendet. Zusätzlich dazu wird der *StepScope* registriert und eine *JobRegistry* erzeugt. Für Testfälle ist unsere Infrastrukturkonfiguration also komplett.

Für das Erzeugen von Jobs und Steps werden nun Builder verwendet, die ebenfalls automatisch im *ApplicationContext* zur Verfügung stehen, wenn *@EnableBatchProcessing* benutzt wird. Unsere Jobkonfiguration landet getrennt von der Infrastruktur in einer eigenen Klasse (Listing 2). Der dort konfigurierte Job liest Datensätze aus einer Datei ein und schreibt sie in eine Datenbank. Der Übersichtlichkeit halber werden einige Elemente verkürzt dargestellt, die komplette Konfiguration findet sich hier [6].

In dieser Konfigurationsklasse werden Reader, Writer, Processor und ein Listener auf die übliche Art und Weise erzeugt. Dabei wird in der Definition des Writers auf die *DataSource* der Infrastrukturkonfiguration zugegriffen. Da wir das Interface referenzieren, ist der Jobkonfiguration egal, wie genau die Implementierung aussieht – perfekte Austauschbarkeit. Trotzdem können wir mit IDE-Mitteln schnell alle möglichen Implementierungen der Methode *dataSource()* finden. Hier sieht man einen großen Vorteil der Java-basierten Konfiguration: Echtes Design mit Typsicherheit, Navigierbarkeit, Delegation und Vererbung sind möglich.

Step und Job werden nun mit entsprechenden Builder-Implementierungen erzeugt. Dafür holt man sich von den zur Verfügung gestellten Builder Factories per *get*-Methode einen Builder, auf dem dann die unterschiedlichsten Operationen durchgeführt werden können. Diese Builder decken die volle Funktionalität des XML-Namespaces ab und sind dementsprechend mächtig. In Listing 2 sehen wir die Basisfunktionen: einen Chunk-basierten Step mit Reader, Processor und

Listing 1

```
@Configuration
@EnableBatchProcessing
public class StandaloneInfrastructureConfiguration implements InfrastructureConfiguration {

  @Bean
  public DataSource dataSource(){
    EmbeddedDatabaseBuilder embeddedDatabaseBuilder = new EmbeddedDatabaseBuilder();
    return embeddedDatabaseBuilder
      .addScript("cp:org/spfw/batch/core/schema-hsqldb.sql")
      .addScript("classpath:schema-partner.sql")
      .setType(EmbeddedDatabaseType.HSQL)
      .build();
  }

}
```

Listing 2

```
@Configuration
public class FlatfileToDbJobConfiguration {

  @Autowired
  private JobBuilderFactory jobBuilders;

  @Autowired
  private StepBuilderFactory stepBuilders;

  @Autowired
  private InfrastructureConfiguration
infrastructureConfiguration;

  @Bean
  public Job flatfileToDbJob(){
    return jobBuilders.get("flatfileToDbJob")
      .listener(protocolListener())
      .start(step())
      .build();
  }

  @Bean
  public Step step(){
    return stepBuilders.get("step")
      .<Partner,Partner>chunk(1)
      .reader(reader())
      .processor(processor())
      .writer(writer())
      .build();
  }

  @Bean
  public FlatFileItemReader<Partner> reader(){
    FlatFileItemReader<Partner> itemReader = new
FlatFileItemReader<Partner>();
    itemReader.setLineMapper(lineMapper());
    itemReader.setResource(new
ClassPathResource("partner-import.csv"));
    return itemReader;
  }

  @Bean
  public LineMapper<Partner> lineMapper(){
    ...
  }

  @Bean
  public ItemProcessor<Partner,Partner> processor(){
    return new LogItemProcessor<Partner>();
  }

  @Bean
  public ItemWriter<Partner> writer(){
    JdbcBatchItemWriter<Partner> itemWriter = new
                    JdbcBatchItemWriter<Partner>();
    itemWriter.setSql(...);
    itemWriter.setDataSource(infrastructureConfigurat
ion.dataSource());
    itemWriter.
setItemSqlParameterSourceProvider(...);
    return itemWriter;
  }

  @Bean
  public ProtocolListener protocolListener(){
    return new ProtocolListener();
  }
}
```

Writer sowie einen normalen Job mit einem *JobExecutionListener* und Referenz auf den Step. Listing 3 zeigt einen typischen JUnit-Integrationstest, der einen Joblauf durchführt.

In Listing 4 sehen wir einen Step, der Skipfunktionalität nutzt. Intern wird dabei übrigens bei Aufruf der Methode *faultTolerant()* die Implementierung des Builders gewechselt. Auf der neuen Implementierung stehen dann alle Methoden bezüglich Skip und Retry zur Verfügung. Dies ist allgemein ein Pattern, das bei den Job und Step Buildern befolgt wird: Es gibt eigene Builder für unterschiedliche Typen von Jobs und Steps, und durch bestimmte Methodenaufrufe wird die Implementierung gewechselt. Der Anwender merkt davon in der Regel nichts.

Zwei sehr wichtige Features in Spring Batch sind der Zugriff auf Jobparameter und der Scope *step* für

Listing 3

```
@ContextConfiguration(classes={StandaloneInfrastructureConfiguration.
                                class, FlatfileToDbJobConfiguration.class})
@RunWith(SpringJUnit4ClassRunner.class)
public class FlatfileToDbJobTests {

  @Autowired
  private JobLauncher jobLauncher;

  @Autowired
  private Job job;

  @Autowired
  private DataSource dataSource;

  private JdbcTemplate jdbcTemplate;

  @Before
  public void setup(){
    jdbcTemplate = new JdbcTemplate(dataSource);
  }

  @Test
  public void testLaunchJob() throws Exception {
    jobLauncher.run(job, new JobParameters());
    assertThat(jdbcTemplate.queryForObject("select count(*) from
                                            partner",Integer.class),is(6));
  }

}
```

Listing 4

```
@Bean
public Step step(){
  return stepBuilders.get("step")
      .<Partner,Partner>chunk(1)
      .reader(reader())
      .processor(processor())
      .writer(writer())
      .listener(logProcessListener())
      .faultTolerant()
      .skipLimit(10)
      .skip(UnknownGenderException.class)
      .listener(logSkipListener())
      .build();
}
```

Listing 5

```
@Bean
@StepScope
public FlatFileItemReader<Partner> reader(@Value("#{jobParameters
                                        [pathToFile]}") String pathToFile){
  FlatFileItemReader<Partner> itemReader = new
                                          FlatFileItemReader<Partner>();
  itemReader.setLineMapper(lineMapper());
  itemReader.setResource(new ClassPathResource(pathToFile));
  return itemReader;
}
```

Listing 6

```
@Configuration
@EnableBatchProcessing
public class WebsphereInfrastructureConfiguration implements
                            BatchConfigurer, InfrastructureConfiguration {

  @Bean
  public DataSource dataSource(){
    try {
      InitialContext initialContext = new InitialContext();
      return (DataSource) initialContext.lookup("java:comp/env/...");
    } catch (NamingException e) {
      throw new RuntimeException("JNDI lookup failed.",e);
    }
  }

  public JobRepository getJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource());
    factory.setTransactionManager(getTransactionManager());
    factory.afterPropertiesSet();
    return  (JobRepository) factory.getObject();
  }

  public PlatformTransactionManager getTransactionManager()
                                                        throws Exception {
    return new WebSphereUowTransactionManager();
  }

  public JobLauncher getJobLauncher() throws Exception {
    SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
    jobLauncher.setJobRepository(getJobRepository());
    jobLauncher.afterPropertiesSet();
    return jobLauncher;
  }

}
```

zustandsbehaftete Komponenten. Der Standard-Scope ist in Spring ja bekanntlich Singleton: Eine Komponente wird beim Start des *ApplicationContexts* erzeugt und lebt dort bis zum Herunterfahren desselben. Das bedeutet, dass die Komponente zustandslos und Thread-safe sein muss, da theoretisch mehrere Jobs gleichzeitig laufen könnten, die die Komponente nutzen. Außerdem ist es nicht möglich, Jobparameter in die Komponente zu injizieren, da diese natürlich erst bei Jobstart zur Verfügung stehen und nicht bei Start des *ApplicationContexts*. Komponenten mit Scope *step* werden dagegen für jede *StepExecution* neu erzeugt und können deshalb einen Zustand haben und auf Jobparameter zugreifen. Dafür muss die Methode, die die Komponente erzeugt, zusätzlich mit *@StepScope* annotiert werden. Dann können per Spring Expression Language und *@Value* Werte aus den Jobparametern als Methodenparameter übergeben werden (Listing 5). Das funktioniert auch für Werte aus dem Job und Step *ExecutionContext* [7].

Zum Abschluss werfen wir noch einen Blick auf eine alternative Implementierung des Interface *InfrastructureConfiguration* für eine Java-EE-Laufzeitumgebung, in diesem Fall ein Websphere Application Server (WAS, Listing 6). Da uns der *DataSourceTransactionManager* nicht reicht, müssen wir nun das Interface *BatchConfigurer* implementieren und die drei Komponenten *PlatformTransactionManager*, *JobRepository* und *JobLauncher* selbst definieren [8]. Dabei nutzen wir das Transaktionsmanagement des WAS und holen uns die *DataSource* per JNDI-Lookup. An unserer Jobkonfiguration müssen wir nichts ändern.

Wer nach dieser Einführung noch tiefer in die Java-basierte Konfiguration in Spring Batch einsteigen möchte, dem kann ich noch meine Blogserie zum Thema ans Herz legen. Neben den Themen, die in diesem Artikel betrachtet werden [6], [7], [8], geht es um die Möglichkeiten, Vererbung zwischen Jobdefinitionen durchzuführen [9], eigene Client-*ApplicationContexts* pro Jobdefinition zu erstellen [10] und Skalierung (Multi-Threading/Partitioning) [11]. Alle Codebeispiele befinden sich in einem öffentlichen GitHub-Repository [12].

Spring-Data-Support

Unter dem Begriff Spring Data sammelt sich eine größere Anzahl unterschiedlicher Projekte, die jeweils den Datenzugriff für eine bestimmte Technologie bereitstellen. So gibt es inzwischen Unterstützung für JPA, MongoDB, Gemfire, Neo4j, Redis, Hadoop und viele mehr [5]. Ein Commons-Projekt bietet übergreifende APIs, soweit diese möglich sind. Dazu gehört eine Repository-Abstraktion.

Spring Batch 2.2 bietet einen *Neo4jItemReader*, einen *MongoItemReader*, einen *Neo4jItemWriter*, einen *MongoItemWriter* und einen *GemfireItemWriter* Out of the Box. Diese basieren jeweils auf Templates aus den entsprechenden Spring-Data-Projekten. Da Spring Batch stark transaktional arbeitet und Datenkonsistenz für Features wie Skip, Retry und Restart sehr wichtig ist [2], stellt sich natürlich die berechtigte Frage, wie diese Komponenten bei der von Spring Batch kontrollierten Transaktion mitspielen. Bei Neo4j und Gemfire ist die Antwort leicht: Beide bringen die Möglichkeit mit, an verteilten XA-Transaktionen teilzunehmen, da entsprechende Treiber vorhanden sind. Wichtig ist natürlich, das auch tatsächlich so zu konfigurieren. In der Regel wird dafür ein Application Server benötigt.

MongoDB bietet diese Möglichkeit nicht. Tatsächlich ist es so, dass hundertprozentige Konsistenz gar nicht erreichbar ist [13]. Der *MongoItemWriter* verzögert das Schreiben der Daten bis kurz vor dem Commit der von Spring Batch kontrollierten Transaktion. Damit ein Fehler vom Framework überhaupt bemerkt wird, müssen im *MongoTemplate* die Property *writeResultChecking* auf *EXCEPTION* und die Property *writeConcern* auf *SAFE* oder größer (am besten *MAJORITY*) stehen. Beachtet man diese Vorgaben nicht, so bemerkt der Batchlauf einen Fehler nicht, läuft erfolgreich durch, hat aber am Ende inkonsistente Daten geschrieben. Eine *Exception* beim Schreiben in die MongoDB schützt uns allerdings noch nicht komplett vor inkonsistenten Daten, da diese immer noch entstehen können, falls mehr als ein Datensatz pro Transaktion geschrieben wird und ein Fehler beim Schreiben des zweiten oder späteren Datensatzes geworfen wird. Der Job sollte dann also immer abbrechen und auch nicht neu gestartet werden. Wenn man diese Besonderheiten beachtet, können alle Features von Spring Batch relativ sicher genutzt werden.

Zusätzlich zu den konkreten Komponenten gibt es noch einen *RepositoryItemReader* und einen *RepositoryItemWriter*. Beide nutzen die Repository-Abstraktion des Spring-Data-Commons-Projekts und erlauben die Verwendung beliebiger Implementierungen aus den Subprojekten. Natürlich sollte auch hier gut auf die Transaktionsfähigkeit der konkreten Implementierung geachtet werden.

Nicht identifizierende Jobparameter

Jobparameter sind Parameter, die beim Start eines Jobs mitgegeben werden und die dann im Verlauf des Jobs genutzt werden können. Das kann beispielsweise ein Tagesdatum sein, mit dem das SQL-Statement zum Ermitteln der Eingabedaten parametrisiert wird. Bis zur jetzigen Version wurde eine *JobInstance* durch alle Jobparameter identifiziert. Wurde also ein Job mit einem bestimmten Parameterset gestartet, wurde anhand der Parameter geprüft, ob ein entsprechender Joblauf schon stattgefunden hat. Falls dieser fehlerhaft war, wurde ein Restart ausgelöst, und falls dieser erfolgreich war, wurde der Jobstart abgewiesen.

Ab Version 2.2 können nun Jobparameter als nicht identifizierend markiert werden. Ein guter Anwendungsfall dafür ist das Commit-Intervall. Dieses kann ja ebenfalls als Jobparameter definiert werden, ist aber ein technischer Parameter und identifiziert den Joblauf nicht fachlich. Wenn man einen fehlerhaften Joblauf mit einem anderen Commit-Intervall neu starten wollte, ging das bisher nicht, da durch die geänderten Parameter die neu zu startende *JobInstance* nicht gefunden wurde. Wenn das Commit-Intervall ein nicht identifizierender Parameter ist, wird es nicht zur Ermittlung der *JobInstance* herangezogen, und ein Neustart ist möglich.

Der kleine Wermutstropfen bei diesem neuen Feature ist die dafür notwendige Änderung des Datenbankschemas: Die Tabelle *BATCH_JOB_PARAMS* fällt weg, dafür kommt die Tabelle *BATCH_JOB_EXECUTION_PARAMS* hinzu. Da alle anderen Tabellen nicht verändert werden, sollte ein Parallelbetrieb verschiedener Versionen problemlos möglich sein. Für die Migration werden Skripte bereitgestellt. Diese befinden sich im Modul *spring-batch-core* im Package *org.springframework.batch.core.migration*.

Weitere Features

Neben den bereits vorgestellten gibt es viele weitere kleinere Features, wie beispielsweise die Unterstützung für AMQP als Datenquelle oder SQLFire als Datenbank. Natürlich wurden auch viele Bugs gefixt. Eine vollständige Auflistung gibt es im Changelog [14].

Fazit

Spring-Data-Support ist interessant für diejenigen, die mit NoSQL-Stores interagieren. Aber das sind im klassischen Batchumfeld eher wenige, auch wenn die Nutzung in Zukunft sicherlich zunehmen wird. Die Möglichkeit, nicht identifizierende Jobparameter zu definieren, ist eine kleine, aber sehr sinnvolle Erweiterung, die vielleicht nicht sehnsüchtig erwartet wurde, aber in Zukunft sicherlich viel genutzt werden wird.

Das wohl interessanteste neue Feature ist die Konfiguration in Java. Der XML-Namespace von Spring Batch ist sehr gut – man kann auf kleinem Raum sehr präzise darstellen, was in einem Job in welcher Reihenfolge passiert. Jobs und Steps über Builder zu erzeugen, wirkt auf den ersten Blick gewöhnungsbedürftig. Ist es jetzt nur Geschmackssache, welchen Konfigurationsstil man verwendet?

Es ist mehr als das. Die Batchverarbeitung findet zum überwiegenden Teil im Enterprise-Bereich statt, beispielsweise bei Versicherungen und Banken. Aufgaben wie Deployment, Infrastruktur, Betrieb, Scheduling, allgemeine Vorgaben für Jobs werden übergreifend gelöst und resultieren in einer relativ dünnen, eigenen Frameworkschicht, die Spring Batch nutzt und die wiederum von allen Batchprojekten im Haus genutzt wird. Und

beim Erstellen von Frameworks hat die Java-basierte Konfiguration diverse Vorteile:

- Echtes Design von Konfigurationen ist möglich. Relationen, Vererbung, Delegation, all das geht. Ein Beispiel? Das im Artikel definierte Interface *InfrastructureConfiguration*. Wir bestimmen darüber, welche Infrastrukturkomponenten wir benötigen, und liefern im Framework noch ein paar Standardimplementierungen. Eine Jobkonfiguration benötigt eine Implementierung, aber dem Job ist es egal, welche das nun ist, ob eine aus dem Framework oder eine vom Nutzer selbst für einen Spezialfall geschriebene. Weitere Beispiele finden sich hier [9].
- Navigierbarkeit ist gegeben. In jeder IDE kann man einfach durch die Konfigurationen navigieren, vor allem auch dann, wenn sich die Konfigurationsklassen nicht in Projekten im Workspace, sondern in eingebundenen JARs befinden. XML-Dateien, die sich in eingebundenen JARs befinden, kann man in Eclipse nicht mal mit Open Resource öffnen; von Navigierbarkeit ganz zu schweigen.

Gerade bei Batchprogrammen ist eine leichte Verständlichkeit elementar für die Wartbarkeit, da an ihnen üblicherweise nicht täglich gearbeitet wird. Häufig werden sie einmal programmiert und laufen dann über Monate und Jahre ohne Änderung. Eine zentrale, Java-basierte Konfiguration punktet in Sachen Wartbarkeit und Nachvollziehbarkeit deutlich.

Zurzeit wird die Version 3.0.0 schon intensiv entwickelt, die dann die Implementierung des JSR 352 (Batch Applications for the Java Platform) enthalten wird. Der JSR 352 ist Thema des nächsten Artikels dieser Reihe. Man darf gespannt sein.

Tobias Flohre arbeitet als Senior Softwareentwickler bei der codecentric AG. Seine Schwerpunkte sind Java-Enterprise-Anwendungen und Architekturen mit JEE/Spring, häufig mit Fokus auf Spring Batch. Er spricht regelmäßig auf Konferenzen und bloggt auf blog.codecentric.de.

tobias.flohre@codecentric.de

Links & Literatur

[1] Schlimm, Niklas; Hinkel, Frank: „Spring Batch 2.0 – Frühling in der Stapelverarbeitung“, in Java Magazin 1.2010

[2] Flohre, Tobias: „Transaktionen in Spring Batch“, in Java Magazin 12.2012: http://www.codecentric.de/kompetenzen/publikationen/transaktionen-in-spring-batch/

[3] http://www.springsource.org/spring-batch

[4] Flohre, Tobias: „Spring komplett ohne XML, geht das? Java-basierte Konfiguration“, in Java Magazin 1.2012: http://www.codecentric.de/kompetenzen/publikationen/spring-komplett-ohne-xml-geht-das-java-basierte-konfiguration/

[5] http://www.springsource.org/spring-data

[6] http://blog.codecentric.de/en/2013/06/spring-batch-2-2-javaconfig-part-1-a-comparison-to-xml/

[7] http://blog.codecentric.de/en/2013/06/spring-batch-2-2-javaconfig-part-2-jobparameters-executioncontext-and-stepscope/

[8] http://blog.codecentric.de/en/2013/06/spring-batch-2-2-javaconfig-part-3-profiles-and-environments/

[9] http://blog.codecentric.de/en/2013/06/spring-batch-2-2-javaconfig-part-4-job-inheritance/

[10] http://blog.codecentric.de/en/2013/06/spring-batch-2-2-javaconfig-part-5-modular-configurations/

[11] http://blog.codecentric.de/en/2013/07/spring-batch-2-2-javaconfig-part-6-partitioning-and-multi-threaded-step/

[12] https://github.com/codecentric/spring-batch-javaconfig

[13] http://www.infoq.com/articles/jepsen

[14] http://static.springsource.org/spring-batch/migration/index.html